characteristic Documentation

Release 15.0.0-dev

Hynek Schlawack

Mar 06, 2017

Contents

Release v15.0.0-dev (*What's new?*).

Warning: Characteristic is **unmaintained**. Please have a look at its successor attrs.

CHAPTER 1

Teaser

```
>>> from characteristic import Attribute, attributes
>>> @attributes(["a", "b"])
... class AClass(object):
...     pass
>>> @attributes(["a", Attribute("b", default_value="abc", instance_of=str)])
... class AnotherClass(object):
...     pass
>>> obj1 = AClass(a=1, b="abc")
>>> obj2 = AnotherClass(a=1, b="abc")
>>> obj3 = AnotherClass(a=1)
>>> AnotherClass(a=1, b=42)
Traceback (most recent call last):
  ...
TypeError: Attribute 'b' must be an instance of 'str'.
>>> print obj1, obj2, obj3
<AClass(a=1, b='abc')> <AnotherClass(a=1, b='abc')> <AnotherClass(a=1, b='abc')>
>>> obj1 == obj2
False
>>> obj2 == obj3
True
```

CHAPTER 2

User's Guide

Why not...

...tuples?

Readability

What makes more sense while debugging:

```
<Point(x=1, y=2)>
```

or:

```
(1, 2)
```

?

Let's add even more ambiguity:

```
<Customer(id=42, reseller=23, first_name="Jane", last_name="John")>
```

or:

```
(42, 23, "Jane", "John")
```

?

Why would you want to write `customer[2]` instead of `customer.first_name`?

Don't get me started when you add nesting. If you've never ran into mysterious tuples you had no idea what the hell they meant while debugging, you're much smarter then I am.

Using proper classes with names and types makes program code much more readable and comprehensible. Especially when trying to grok a new piece of software or returning to old code after several months.

Extendability

Imagine you have a function that takes or returns a tuple. Especially if you use tuple unpacking (eg. `x, y = get_point()`), adding additional data means that you have to change the invocation of that function *everywhere*.

Adding an attribute to a class concerns only those who actually care about that attribute.

. . . namedtuples?

The difference between namedtuples and classes decorated by `characteristic` is that the latter are type-sensitive and less typing aside regular classes:

```
>>> from characteristic import Attribute, attributes
>>> @attributes([Attribute("a", instance_of=int)])
... class C1(object):
...     def __init__(self):
...         if self.a >= 5:
...             raise ValueError("'a' must be smaller than 5!")
...     def print_a(self):
...         print self.a
>>> @attributes([Attribute("a", instance_of=int)])
... class C2(object):
...     pass
>>> c1 = C1(a=1)
>>> c2 = C2(a=1)
>>> c1.a == c2.a
True
>>> c1 == c2
False
>>> c1.print_a()
1
>>> C1(a=5)
Traceback (most recent call last):
  ...
ValueError: 'a' must be smaller than 5!
```

. . . while namedtuple's purpose is *explicitly* to behave like tuples:

```
>>> from collections import namedtuple
>>> NT1 = namedtuple("NT1", "a")
>>> NT2 = namedtuple("NT2", "b")
>>> t1 = NT1._make([1,])
>>> t2 = NT2._make([1,])
>>> t1 == t2 == (1,)
True
```

This can easily lead to surprising and unintended behaviors.

Other than that, `characteristic` also adds nifty features like type checks or default values.

. . . hand-written classes?

While I'm a fan of all things artisanal, writing the same nine methods all over again doesn't qualify for me. I usually manage to get some typos inside and there's simply more code that can break and thus has to be tested.

To bring it into perspective, the equivalent of

```
>>> @attributes(["a", "b"])
... class SmartClass(object):
...     pass
>>> SmartClass(a=1, b=2)
<SmartClass(a=1, b=2)>
```

is

```
>>> class ArtisanalClass(object):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...
...     def __repr__(self):
...         return "<ArtisanalClass(a={}, b={})>".format(self.a, self.b)
...
...     def __eq__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) == (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __ne__(self, other):
...         result = self.__eq__(other)
...         if result is NotImplemented:
...             return NotImplemented
...         else:
...             return not result
...
...     def __lt__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) < (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __le__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) <= (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __gt__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) > (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __ge__(self, other):
...         if other.__class__ is self.__class__:
...             return (self.a, self.b) >= (other.a, other.b)
...         else:
...             return NotImplemented
...
...     def __hash__(self):
...         return hash((self.a, self.b))
>>> ArtisanalClass(a=1, b=2)
<ArtisanalClass(a=1, b=2)>
```

which is quite a mouthful and it doesn't even use any of `characteristic`'s more advanced features like type checks or default values Also: no tests whatsoever. And who will guarantee you, that you don't accidentally flip the < in your tenth implementation of `__gt__`?

If you don't care and like typing, I'm not gonna stop you. But if you ever get sick of the repetitiveness, `characteristic` will be waiting for you. :)

Examples

`@attributes` together with the definition of the attributes using class attributes enhances your class by:

- a nice `__repr__`,
- comparison methods that compare instances as if they were tuples of their attributes,
- and an initializer that uses the keyword arguments to initialize the specified attributes before running the class' own initializer (you just write the validator if you need anything more than type checks!).

```
>>> from characteristic import Attribute, attributes
>>> @attributes(["a", "b"])
... class C(object):
...     pass
>>> obj1 = C(a=1, b="abc")
>>> obj1
<C(a=1, b='abc')>
>>> obj2 = C(a=2, b="abc")
>>> obj1 == obj2
False
>>> obj1 < obj2
True
>>> obj3 = C(a=1, b="bca")
>>> obj3 > obj1
True
```

To offer more power and possibilities, `characteristic` comes with a distinct class to define attributes: *`Attribute`*. It allows for things like default values for certain attributes, making them optional when `characteristic`'s generated initializer is used:

```
>>> @attributes(["a", "b", Attribute("c", default_value=42)])
... class CWithDefaults(object):
...     pass
>>> obj4 = CWithDefaults(a=1, b=2)
>>> obj4.characteristic_attributes
[<Attribute(name='a', exclude_from_cmp=False, exclude_from_init=False, exclude_from_
↪repr=False, exclude_from_immutable=False, default_value=NOTHING, default_
↪factory=None, instance_of=None, init_aliaser=None)>, <Attribute(name='b', exclude_
↪from_cmp=False, exclude_from_init=False, exclude_from_repr=False, exclude_from_
↪immutable=False, default_value=NOTHING, default_factory=None, instance_of=None,␣
↪init_aliaser=None)>, <Attribute(name='c', exclude_from_cmp=False, exclude_from_
↪init=False, exclude_from_repr=False, exclude_from_immutable=False, default_value=42,
↪ default_factory=None, instance_of=None, init_aliaser=<function strip_leading_
↪underscores at ...>)>]
>>> obj5 = CWithDefaults(a=1, b=2, c=42)
>>> obj4 == obj5
True
```

`characteristic` also offers factories for default values of complex types:

```
>>> @attributes([Attribute("a", default_factory=list),
...              Attribute("b", default_factory=dict)])
... class CWithDefaultFactory(object):
...     pass
>>> obj6 = CWithDefaultFactory()
>>> obj6
<CWithDefaultFactory(a=[], b={})>
>>> obj7 = CWithDefaultFactory()
>>> obj7
<CWithDefaultFactory(a=[], b={})>
>>> obj6 == obj7
True
>>> obj6.a is obj7.a
False
>>> obj6.b is obj7.b
False
```

You can also exclude certain attributes from certain decorators:

```
>>> @attributes(["host", "user",
...              Attribute("password", exclude_from_repr=True),
...              Attribute("_connection", exclude_from_init=True)])
... class DB(object):
...     _connection = None
...     def connect(self):
...         self._connection = "not really a connection"
>>> db = DB(host="localhost", user="dba", password="secret")
>>> db.connect()
>>> db
<DB(host='localhost', user='dba', _connection='not really a connection')>
```

Immutable data structures are amazing! Guess what `characteristic` supports?

```
>>> @attributes([Attribute("a")], apply_immutable=True)
... class ImmutableClass(object):
...     pass
>>> ic = ImmutableClass(a=42)
>>> ic.a
42
>>> ic.a = 43
Traceback (most recent call last):
 ...
AttributeError: Attribute 'a' of class 'ImmutableClass' is immutable.
>>> @attributes([Attribute("a")], apply_immutable=True)
... class AnotherImmutableClass(object):
...     def __init__(self):
...         self.a *= 2
>>> ic2 = AnotherImmutableClass(a=21)
>>> ic2.a
42
>>> ic.a = 43
Traceback (most recent call last):
 ...
AttributeError: Attribute 'a' of class 'AnotherImmutableClass' is immutable.
```

You know what else is amazing? Type checks!

```
>>> @attributes([Attribute("a", instance_of=int)])
... class TypeCheckedClass(object):
...     pass
>>> TypeCheckedClass(a="totally not an int")
Traceback (most recent call last):
 ...
TypeError: Attribute 'a' must be an instance of 'int'.
```

And if you want your classes to have certain attributes private, `characteristic` will keep your keyword arguments clean if not told otherwise*[0]:

```
>>> @attributes([Attribute("_private")])
... class CWithPrivateAttribute(object):
...     pass
>>> obj8 = CWithPrivateAttribute(private=42)
>>> obj8._private
42
>>> @attributes([Attribute("_private", init_aliaser=None)])
... class CWithPrivateAttributeNoAliasing(object):
...     pass
>>> obj9 = CWithPrivateAttributeNoAliasing(_private=42)
>>> obj9._private
42
```

API

`characteristic` consists of several class decorators that add features to your classes. There are four that add *one* feature each to your class. And then there's the helper `@attributes` that combines them all into one decorator so you don't have to repeat the attribute list multiple times.

Generally the decorators take a list of attributes as their first positional argument. This list can consists of either native strings*[0] for simple cases or instances of *Attribute* that allow for more customization of `characteristic`'s behavior.

The easiest way to get started is to have a look at the *Examples* to get a feeling for `characteristic` and return later for details!

Note: Every argument except for `attrs` for decorators and `name` for *Attribute* is a **keyword argument**. Their positions are coincidental and not guaranteed to remain stable.

`characteristic.`**`attributes`**(*attrs*, *apply_with_cmp=True*, *apply_with_init=True*, *apply_with_repr=True*, *apply_immutable=False*, *store_attributes=<function _default_store_attributes>*, ***kw*)

A convenience class decorator that allows to *selectively* apply *with_cmp()*, *with_repr()*, *with_init()*, and *immutable()* to avoid code duplication.

Parameters

- **attrs** (`list` of `str` or *Attribute*s.) – Attributes to work with.
- **apply_with_cmp** (*bool*) – Apply *with_cmp()*.
- **apply_with_init** (*bool*) – Apply *with_init()*.

[0] This works *only* for attributes defined using the *Attribute* class.
[0] Byte strings on Python 2 and Unicode strings on Python 3.

- **apply_with_repr** (*bool*) – Apply *with_repr()*.
- **apply_immutable** (*bool*) – Apply *immutable()*. The only one that is off by default.
- **store_attributes** (*callable*) – Store the given `attrs` on the class. Should accept two arguments, the class and the attributes, in that order. Note that attributes passed in will always be instances of *Attribute*, (so simple string attributes will already have been converted). By default if unprovided, attributes are stored in a `characteristic_attributes` attribute on the class.

Raises **ValueError** – If both *defaults* and an instance of *Attribute* has been passed.

New in version 14.0: Added possibility to pass instances of *Attribute* in `attrs`.

New in version 14.0: Added `apply_*`.

New in version 14.2: Added `store_attributes`.

Deprecated since version 14.0: Use *Attribute* instead of `defaults`.

Parameters **defaults** (`dict` or `None`) – Default values if attributes are omitted on instantiation.

Deprecated since version 14.0: Use `apply_with_init` instead of `create_init`. Until removal, if *either* if *False*, `with_init` is not applied.

Parameters **create_init** (*bool*) – Apply *with_init()*.

`characteristic.`**with_repr**(*attrs*)

A class decorator that adds a human readable `__repr__` method to your class using *attrs*.

Parameters **attrs** (`list` of `str` or *Attribute*s.) – Attributes to work with.

```
>>> from characteristic import with_repr
>>> @with_repr(["a", "b"])
... class RClass(object):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
>>> c = RClass(42, "abc")
>>> print c
<RClass(a=42, b='abc')>
```

`characteristic.`**with_cmp**(*attrs*)

A class decorator that adds comparison methods and a hashing method based on *attrs*.

For that, each class is treated like a `tuple` of the values of *attrs*, so objectA == objectB -> True and objectA.__hash__() == objectB.__hash() -> True iff objectA's tuple of *attrs* == objectB's tuple of *attrs*. But only instances of *identical* classes are compared!

Parameters **attrs** (`list` of `str` or *Attribute*s.) – Attributes to work with.

```
>>> from characteristic import with_cmp
>>> @with_cmp(["a", "b"])
... class CClass(object):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
>>> o1 = CClass(1, "abc")
>>> o2 = CClass(1, "abc")
>>> o1 == o2  # o1.a == o2.a and o1.b == o2.b
True
>>> o1.c = 23
>>> o2.c = 42
```

```
>>> o1 == o2  # attributes that are not passed to with_cmp are ignored
True
>>> o3 = CClass(2, "abc")
>>> o1 < o3  # because 1 < 2
True
>>> o4 = CClass(1, "bca")
>>> o1 < o4  # o1.a == o4.a, but o1.b < o4.b
True
```

characteristic.**with_init**(*attrs*, ***kw*)

A class decorator that wraps the `__init__` method of a class and sets *attrs* using passed *keyword arguments* before calling the original `__init__`.

Those keyword arguments that are used, are removed from the *kwargs* that is passed into your original `__init__`. Optionally, a dictionary of default values for some of *attrs* can be passed too.

Attributes that are defined using *Attribute* and start with underscores will get them stripped for the initializer arguments by default (this behavior is changeable on per-attribute basis when instantiating *Attribute*.

Parameters **attrs** (`list` of `str` or *Attribute*s.) – Attributes to work with.

Raises

- **ValueError** – If the value for a non-optional attribute hasn't been passed as a keyword argument.
- **ValueError** – If both *defaults* and an instance of *Attribute* has been passed.

Deprecated since version 14.0: Use *Attribute* instead of `defaults`.

Parameters **defaults** (`dict` or `None`) – Default values if attributes are omitted on instantiation.

```
>>> from characteristic import with_init, Attribute
>>> @with_init(["a",
...             Attribute("b", default_factory=lambda: 2),
...             Attribute("_c")])
... class IClass(object):
...     def __init__(self):
...         if self.b != 2:
...             raise ValueError("'b' must be 2!")
>>> o1 = IClass(a=1, b=2, c=3)
>>> o2 = IClass(a=1, c=3)
>>> o1._c
3
>>> o1.a == o2.a
True
>>> o1.b == o2.b
True
>>> IClass()
Traceback (most recent call last):
  ...
ValueError: Missing keyword value for 'a'.
>>> IClass(a=1, b=3)  # the custom __init__ is called after the attributes are␣
↪initialized
Traceback (most recent call last):
  ...
ValueError: 'b' must be 2!
```

Note: The generated initializer explicitly does *not* support positional arguments. Those are *always* passed

to the existing `__init__` unaltered. Used keyword arguments will *not* be passed to the original `__init__` method and have to be accessed on the class (i.e. `self.a`).

`characteristic.`**`immutable`**(*attrs*)

Class decorator that makes *attrs* of a class immutable.

That means that *attrs* can only be set from an initializer. If anyone else tries to set one of them, an `AttributeError` is raised.

New in version 14.0.

```
>>> from characteristic import immutable
>>> @immutable([Attribute("foo")])
... class ImmutableClass(object):
...     foo = "bar"
>>> ic = ImmutableClass()
>>> ic.foo
'bar'
>>> ic.foo = "not bar"
Traceback (most recent call last):
  ...
AttributeError: Attribute 'foo' of class 'ImmutableClass' is immutable.
```

Please note, that that doesn't mean that the attributes themselves are immutable too:

```
>>> @immutable(["foo"])
... class C(object):
...     foo = []
>>> i = C()
>>> i.foo = [42]
Traceback (most recent call last):
 ...
AttributeError: Attribute 'foo' of class 'C' is immutable.
>>> i.foo.append(42)
>>> i.foo
[42]
```

class `characteristic.`**`Attribute`**(*name*, *exclude_from_cmp=False*, *exclude_from_init=False*, *exclude_from_repr=False*, *exclude_from_immutable=False*, *default_value=NOTHING*, *default_factory=None*, *instance_of=None*, *init_aliaser=<function strip_leading_underscores>*)

A representation of an attribute.

In the simplest case, it only consists of a name but more advanced properties like default values are possible too.

All attributes on the Attribute class are *read-only*.

Parameters

- **`name`** (*`str`*) – Name of the attribute.
- **`exclude_from_cmp`** (*`bool`*) – Ignore attribute in *`with_cmp()`*.
- **`exclude_from_init`** (*`bool`*) – Ignore attribute in *`with_init()`*.
- **`exclude_from_repr`** (*`bool`*) – Ignore attribute in *`with_repr()`*.
- **`exclude_from_immutable`** (*`bool`*) – Ignore attribute in *`immutable()`*.
- **`default_value`** – A value that is used whenever this attribute isn't passed as an keyword argument to a class that is decorated using *`with_init()`* (or *`attributes()`*

with `apply_with_init=True`).

Therefore, setting this makes an attribute *optional*.

Since a default value of *None* would be ambiguous, a special sentinel *NOTHING* is used. Passing it means the lack of a default value.

- **default_factory** (*callable*) – A factory that is used for generating default values whenever this attribute isn't passed as an keyword argument to a class that is decorated using *with_init()* (or *attributes()* with `apply_with_init=True`).

  Therefore, setting this makes an attribute *optional*.

- **instance_of** (*type*) – If used together with *with_init()* (or *attributes()* with `apply_with_init=True`), the passed value is checked whether it's an instance of the type passed here. The initializer then raises `TypeError` on mismatch.

- **init_aliaser** (*callable*) – A callable that is invoked with the name of the attribute and whose return value is used as the keyword argument name for the `__init__` created by *with_init()* (or *attributes()* with `apply_with_init=True`). Uses *strip_leading_underscores()* by default to change `_foo` to `foo`. Set to `None` to disable aliasing.

Raises **ValueError** – If both `default_value` and `default_factory` have been passed.

New in version 14.0.

`characteristic.`**strip_leading_underscores**(*attribute_name*)

Strip leading underscores from *attribute_name*.

Used by default by the `init_aliaser` argument of *Attribute*.

Parameters **attribute_name** (*str*) – The original attribute name to mangle.

Return type str

```
>>> from characteristic import strip_leading_underscores
>>> strip_leading_underscores("_foo")
'foo'
>>> strip_leading_underscores("__bar")
'bar'
>>> strip_leading_underscores("___qux")
'qux'
```

`characteristic.`**NOTHING** = NOTHING

Sentinel to indicate the lack of a value when `None` is ambiguous.

New in version 14.0.

Project Information

License and Hall of Fame

`characteristic` is licensed under the permissive MIT license. The full license text can be also found in the source code repository.

Authors

`characteristic` is written and maintained by Hynek Schlawack.

The development is kindly supported by Variomedia AG.

It's inspired by Twisted's FancyEqMixin but is implemented using class decorators because sub-classing is bad for you, m'kay?

The following folks helped forming `characteristic` into what it is now:

- Adam Dangoor
- Glyph
- Itamar Turner-Trauring
- Jean-Paul Calderone
- Julian Berman
- Richard Wall
- Tom Prince

How To Contribute

Every open source project lives from the generous help by contributors that sacrifice their time and `characteristic` is no different.

To make participation as pleasant as possible, this project adheres to the Code of Conduct by the Python Software Foundation.

Here are a few guidelines to get you started:

- Add yourself to the AUTHORS.rst file in an alphabetical fashion. Every contribution is valuable and shall be credited.
- If your change is noteworthy, add an entry to the changelog.
- No contribution is too small; please submit as many fixes for typos and grammar bloopers as you can!
- Don't *ever* break backward compatibility. If it ever *has* to happen for higher reasons, `characteristic` will follow the proven procedures of the Twisted project.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation won't be merged. If a feature is not tested or documented, it doesn't exist.
- Obey PEP 8 and PEP 257.
- Write good commit messages.

Note: If you have something great but aren't sure whether it adheres – or even can adhere – to the rules above: **please submit a pull request anyway**!

In the best case, we can mold it into something, in the worst case the pull request gets politely closed. There's absolutely nothing to fear.

Thank you for considering to contribute to `characteristic`! If you have any question or concerns, feel free to reach out to me.

Changelog

Versions are year-based with a strict backwards-compatibility policy. The third digit is only for regressions.

14.3.0 (2014-12-19)

Backward-incompatible changes:

none

Deprecations:

none

Changes:

- All decorators now gracefully accept empty attribute lists. [22].

14.2.0 (2014-10-30)

Backward-incompatible changes:

none

Deprecations:

none

Changes:

- Attributes set by *characteristic.attributes()* are now stored on the class as well. [20]
- `__init__` methods that are created by *characteristic.with_init()* are now generated on the fly and optimized for each class. [9]

14.1.0 (2014-08-22)

Backward-incompatible changes:

none

Deprecations:

none

Changes:

- Fix stray deprecation warnings.
- Don't rely on warnings being switched on by command line. [17]

14.0.0 (2014-08-21)

Backward-incompatible changes:

none

Deprecations:

- The `defaults` argument of *`with_init()`* and *`attributes()`* has been deprecated in favor of the new explicit *`Attribute`* class and it's superior `default_value` and `default_factory` arguments.
- The `create_init` argument of *`attributes()`* has been deprecated in favor of the new `apply_with_init` argument for the sake of consistency.

Changes:

- Switch to a year-based version scheme.
- Add *`immutable()`* to make certain attributes of classes immutable. Also add `apply_immutable` argument to *`attributes()`*. [14]
- Add explicit *`Attribute`* class and use it for default factories. [8]
- Add aliasing of private attributes for *`with_init()`*'s initializer when used together with *`Attribute`*. Allow for custom aliasing via a callable. [6, 13]
- Add type checks to *`with_init()`*'s initializer. [12]
- Add possibility to hand-pick which decorators are applied from within *`attributes()`*.
- Add possibility to exclude single attributes from certain decorators.

0.1.0 (2014-05-11)

- Initial release.

CHAPTER 3

Indices and tables

- genindex
- search

Index